LECTURE 6
LINEAR LIST

SEHH2239 Data Structures

Linear Lists

After completing this lesson, you should be able to do the following:

- Use Pointer to represent Linear List
- Implement Linear List Structure and Operations with Python pointer and List

LINEAR LIST ADT

Abstract Data Type (ADT)

- Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.
- The definition of ADT only mentions **what operations** are to be performed but **not how** these operations will be implemented.
- It is called “abstract” because it gives an implementation-independent view.
  - The process of providing only the essentials and hiding the details is known as abstraction.
  - It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.

https://www.geeksforgeeks.org/abstract-data-types/

Linear Lists

$L = (e_0, e_1, e_2, e_3, \ldots, e_{n-1})$

- relationships
- where e_i denotes a list element and list size is n
- $n >= 0$ is finite
- e_0 is the zero'th (or front) element
- e_{n-1} is the last element
- e_i immediately precedes e_{i+1}

Linear List Examples

- Students in SEHH2239 =
  (Jack, Jill, Abe, Henry, Mary, …, Judy)
- Quizzes in SEHH2239 =
  (quiz1, quiz2, quiz3)
- Days of Week = (S, M, T, W, Th, F, Sa)
- Months = (Jan, Feb, Mar, Apr, …, Nov, Dec)

Linear List Abstract Data Type

AbstractDataType *LinearList*

Operations:

isEmpty(): return true iff the list is empty, false otherwise

size(): return the size of the list

get(i): return the *i* th element of the list

indexOf(el): return the index of the first occurrence of el in the list, return -1 if *x* is not in the list

remove(index): remove and return the *index*th element, elements with higher index have their index reduced by 1

removeNode(Removekey): remove the node with given element elements with higher index have their index reduced by 1

add(theIndex, x): insert x as the *index*th element, elements with theIndex >= *index* have their index increased by 1

addAtHead(x): inert the x at the beginning

addAtTail(x): inert the x at the end

listprint(): print the linked list

Linear List Operations—size()

- determine list size

$L = (a,b,c,d,e)$

size = 5

Linear List Operations—get(theIndex)

- get element with given index

$$L = (a,b,c,d,e)$$

- *get(0)* = *a*
- *get(2)* = *c*
- *get(4)* = *e*
- *get(-1)* = error
- *get(9)* = error

Linear List Operations—indexOf(theElement)

- determine the index of an element

$$L = (a,b,d,b,a)$$

- *indexOf(d) = 2*
- *indexOf(a) = 0*
- *indexOf(z) = -1*

Linear List Operations—remove(theIndex)

- remove and return element with given index

$$L = (a,b,c,d,e,f,g)$$

- *remove(2)* returns *c*
- and *L* becomes *(a,b,d,e,f,g)*

index of *d,e,f,* and *g* decrease by *1*

Linear List Operations—remove(theIndex)

- remove and return element with given index

$$L = (a,b,c,d,e,f,g)$$

- *remove(-1)* => error
- *remove(20)* => error

Linear List Operations—remove(theElement)

- remove the first occurrence of the specified element.

$$L = (a,b,c,d,e,f,g)$$

- *remove(c)* and *L* becomes *(a,b,d,e,f,g)*

  index of *d*,*e*,*f*, and *g* decrease by *1*

- *remove(h)* => no element removed

Linear List Operations—add(theIndex, theElement)

- add an element so that the new element has a specified index

$$L = (a,b,c,d,e,f,g)$$

- *add(0,h) => L = (h,a,b,c,d,e,f,g)*

  index of *a*,*b*,*c*,*d*,*e*,*f*, and *g* increase by *1*

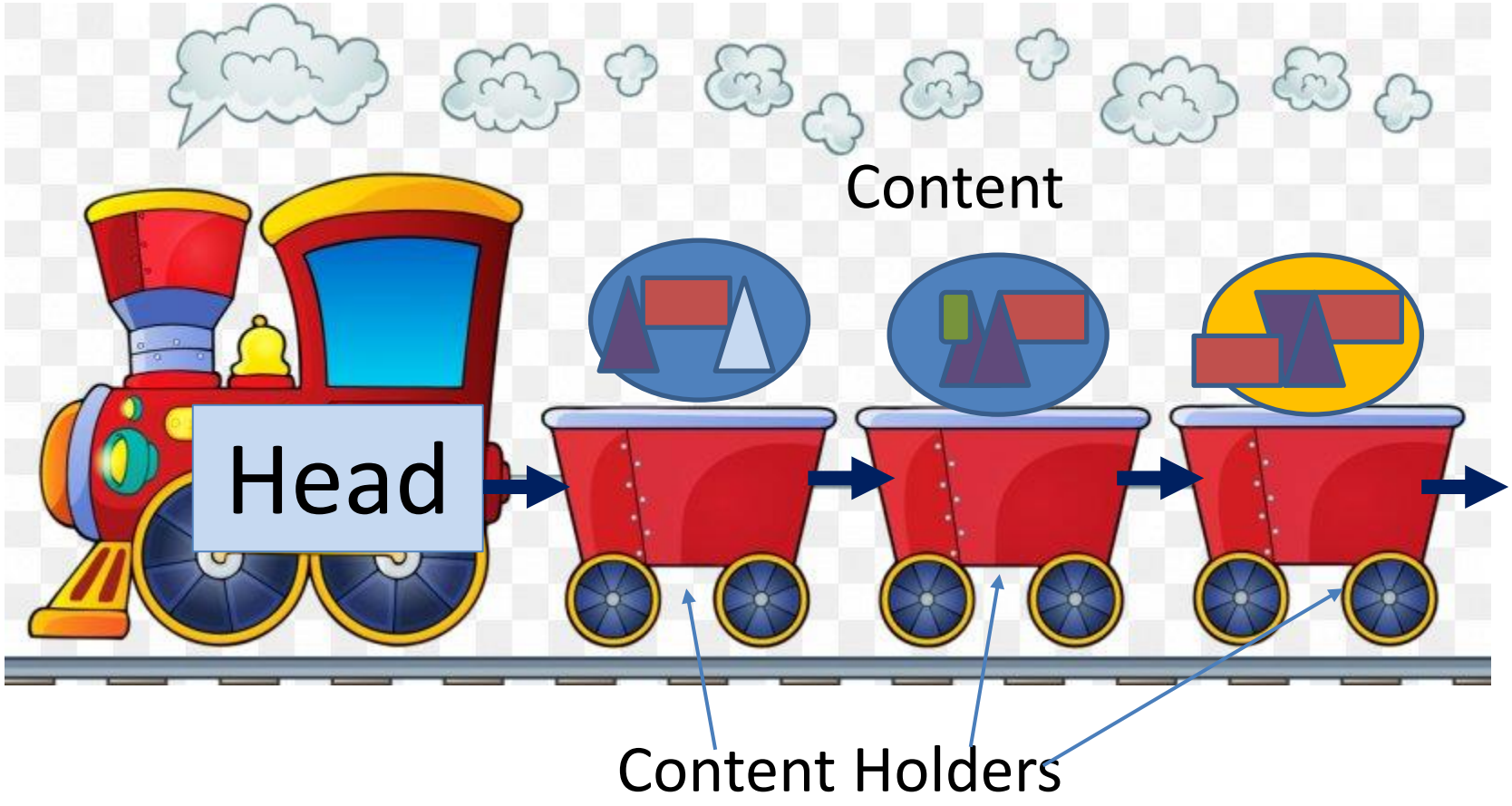
Linear List Operations—add(theIndex, theElement)

L = (a,b,c,d,e,f,g)

- *add(2,h)* => *L = (a,b,h,c,d,e,f,g)*
  index of *c,d,e,f,* and *g* increase by *1*
- *add(10,h)* => error
- *add(-6,h)* => error

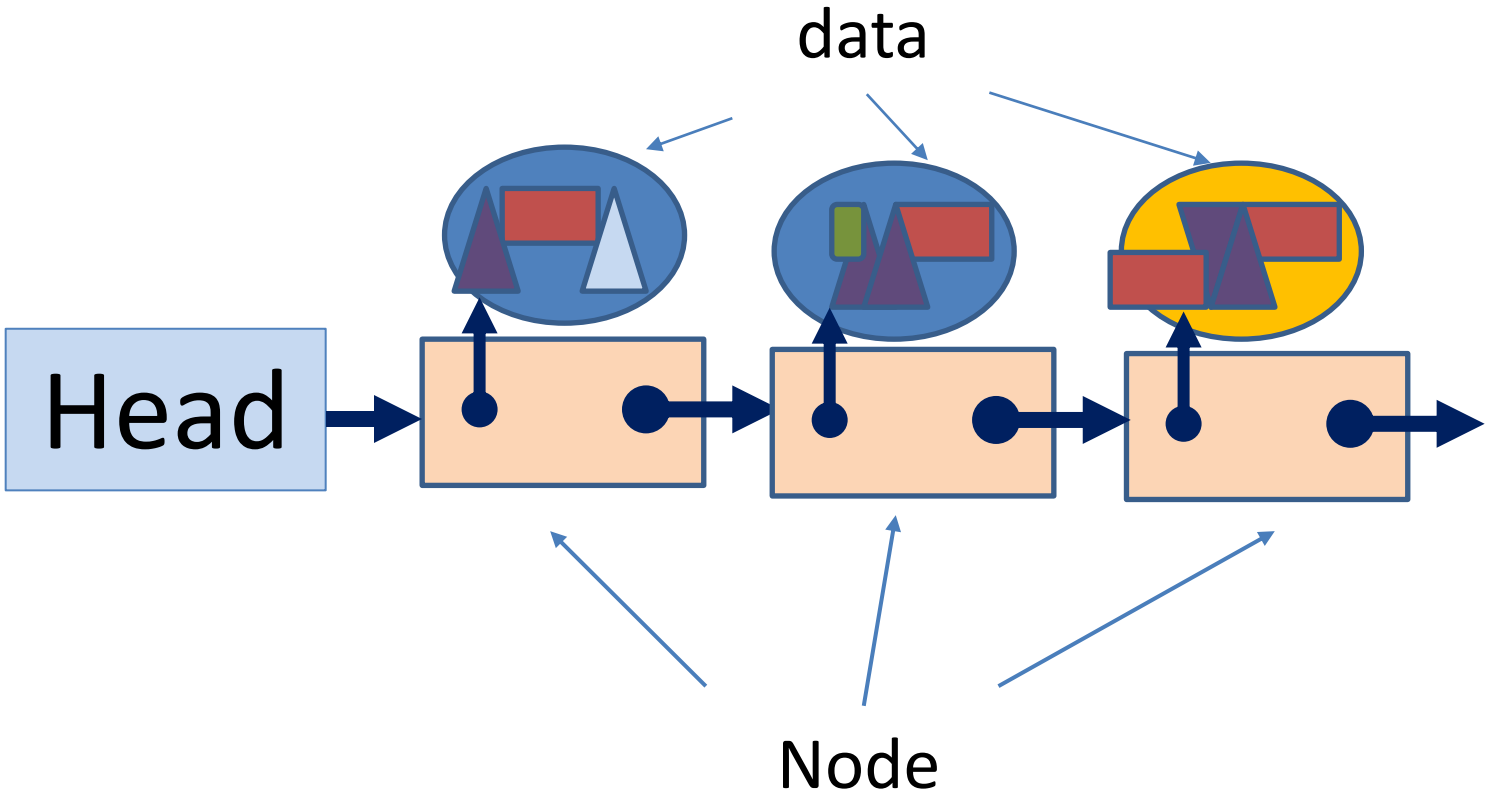
Linear List Implementation – Singly Linked List

Consider a train

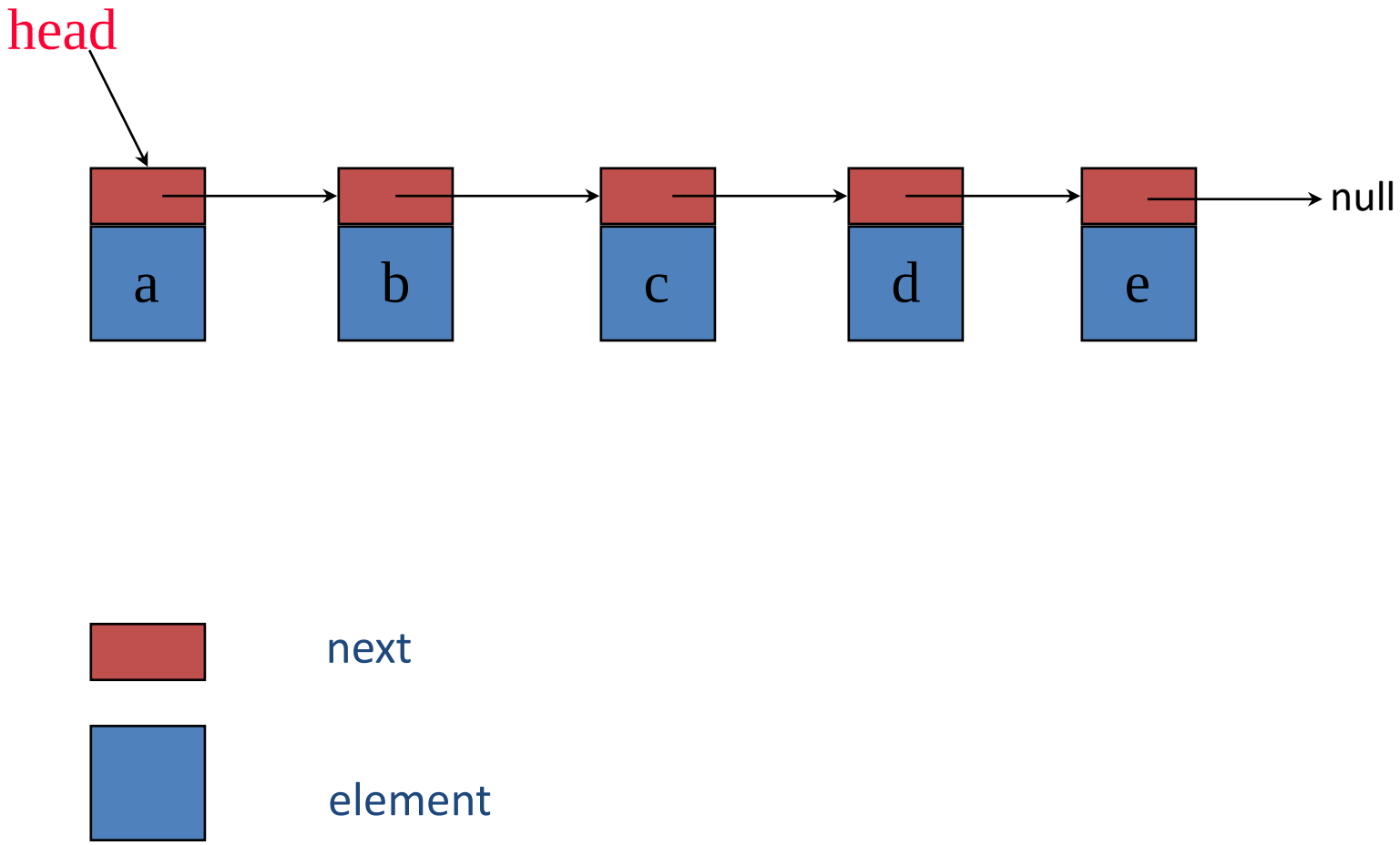


https://favpng.com/png_view/train-train-rail-transport-clip-art-image-png/xu2Tnfr3

A chain of nodes



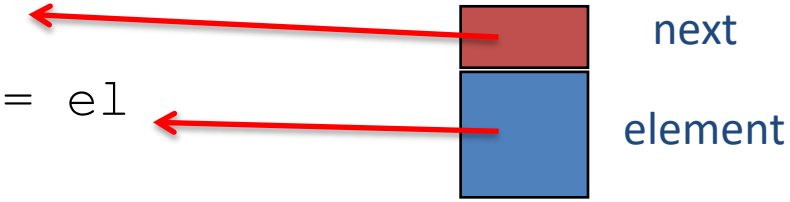
The Class Node



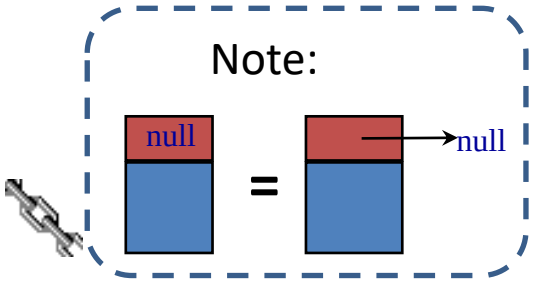
size = number of elements

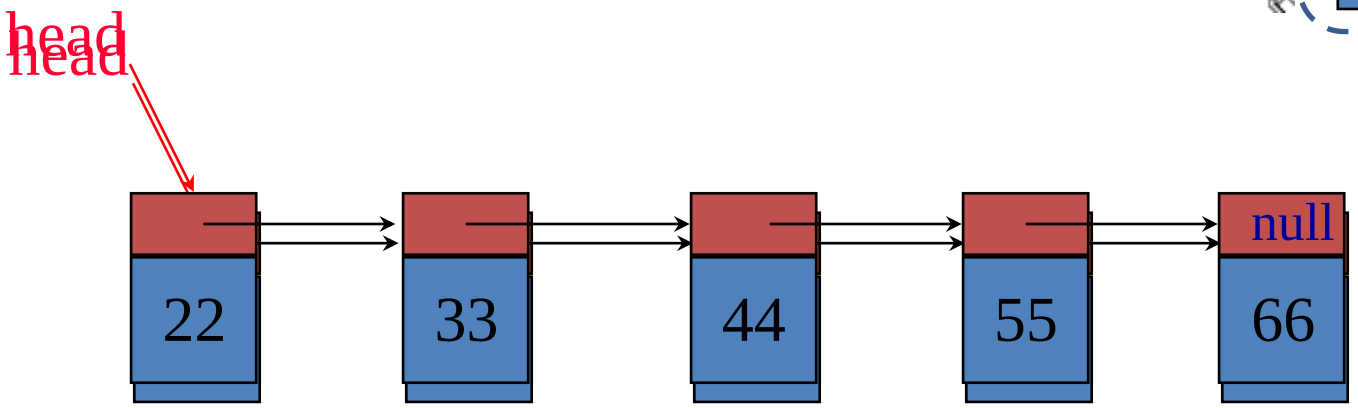
Creation of Linked list

```
class Node:
    def __init__(self, el = None, n = None):
        self.next = n
        self.element = el

class SLinkedList:
   def __init__(self):
       self.head = None
```



LinkedList





To implement this chain:

Method 1:

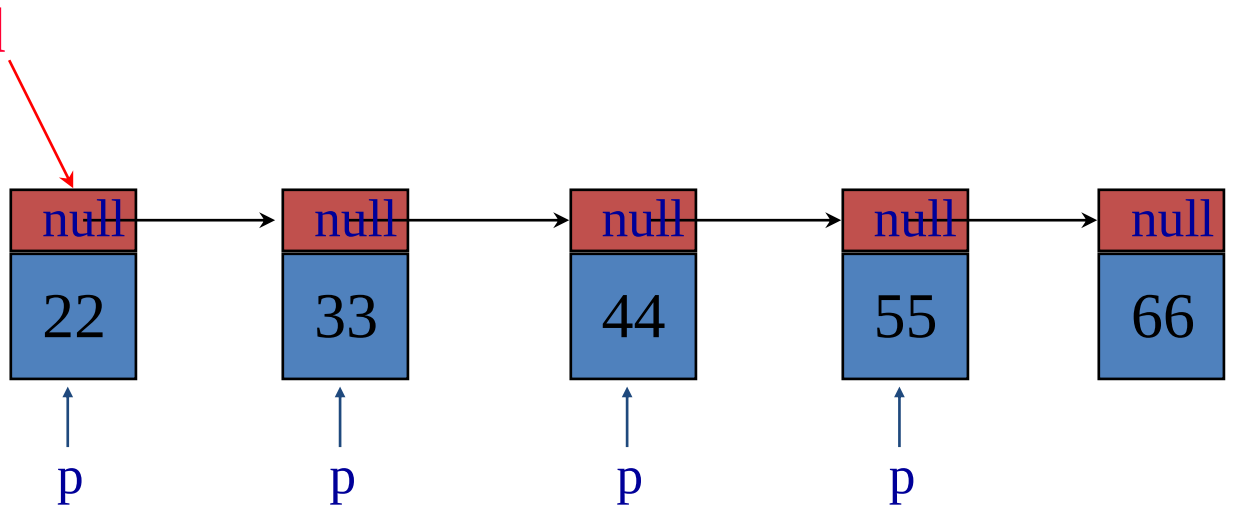
```
list1 = SLinkedList()
list1.head = Node(22)

list1.head.next = Node(33)

list1.head.next.next = Node(44)

list1.head.next.next.next = Node(55)

list1.head.next.next.next.next = Node(66)
```

```
class Node:
    def __init__(self, el = None, n = None):
        self.next = n
        self.element = el

class SLinkedList:
   def __init__(self):
      self.head = None

   def listprint(self):
      printval = self.head
      while printval is not None:
         print (printval.element)
         printval = printval.next

list1 = SLinkedList()
list1.head = Node(22)

# Link first Node to second node
list1.head.next = Node(33)

# Link second Node to third node.. so on so fore
list1.head.next.next = Node(44)
list1.head.next.next.next = Node(55)
list1.head.next.next.next.next = Node(66)

list1.listprint()
```

```
22
33
44
55
66
```

SLLEx1.py

Linked List

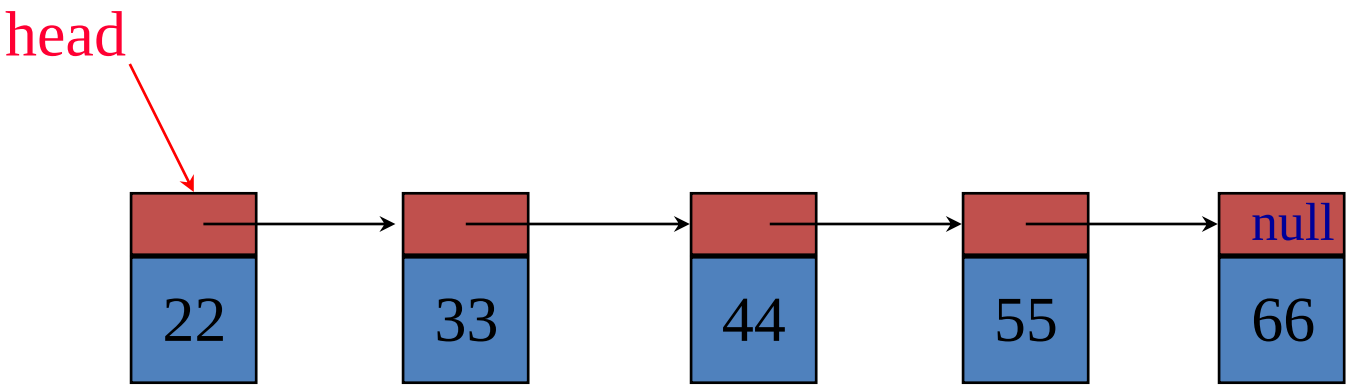


Method 2:

```
list2 = SLinkedList()
list2.head = Node(22)
p = list2.head
p.next = Node(33)
p = p.next;
p.next = Node(44)
p = p.next;
p.next = Node(55)
p = p.next;
p.next = Node(66)
```

SLLEx1.py

Linked List



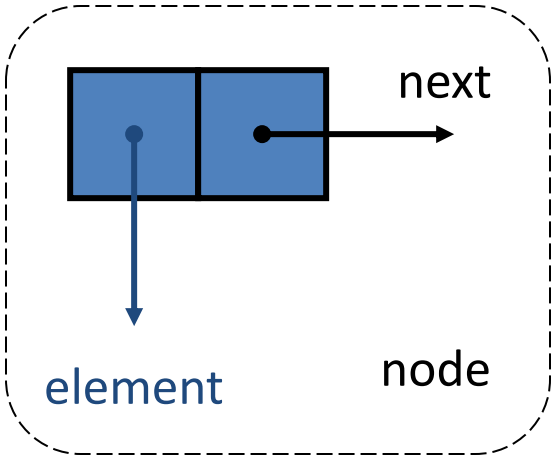
Method 3:

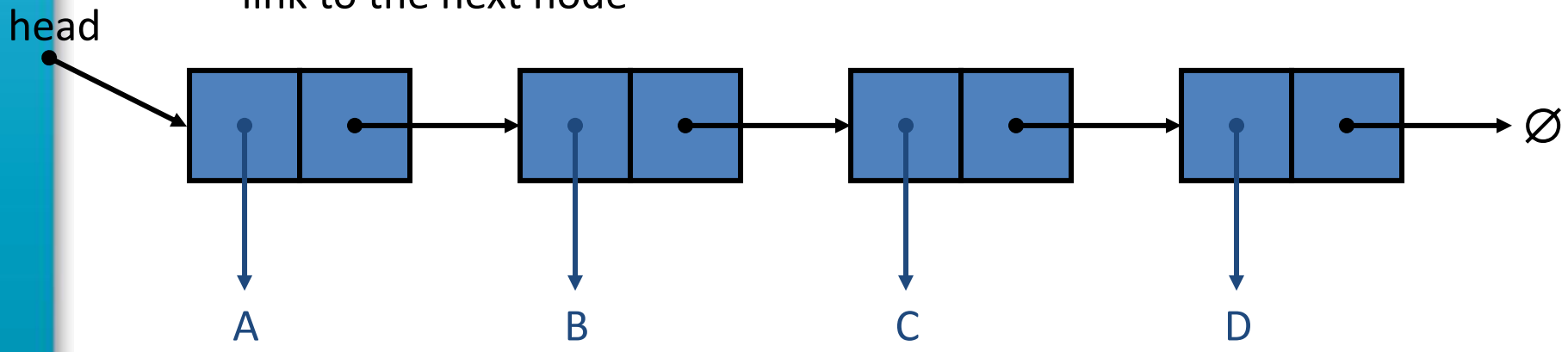
```
array1 = array('i', [33,44,55,66])
list3 = SLinkedList()
list3.head = Node(22)
p = list3.head

for x in array1:
    p.next = Node(x)
    p = p.next
```

SLLEx1.py

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node





The Class SLinkedList

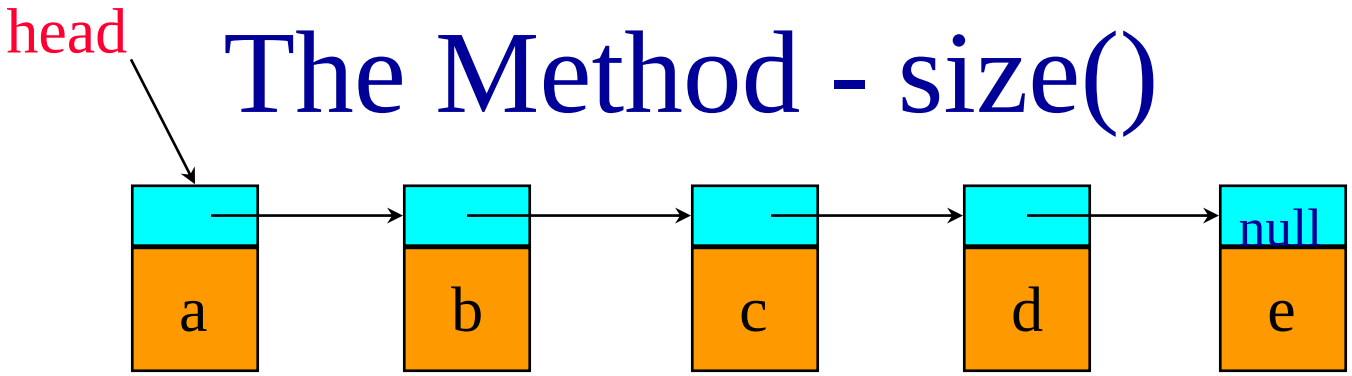
```
class Node:
    def __init__(self, el = None, n = None):
        self.next = n
        self.element = el

  class SLinkedList:
   def __init__(self):
        self.head = None
```

SLinkedList.py

The Method – isEmpty() 

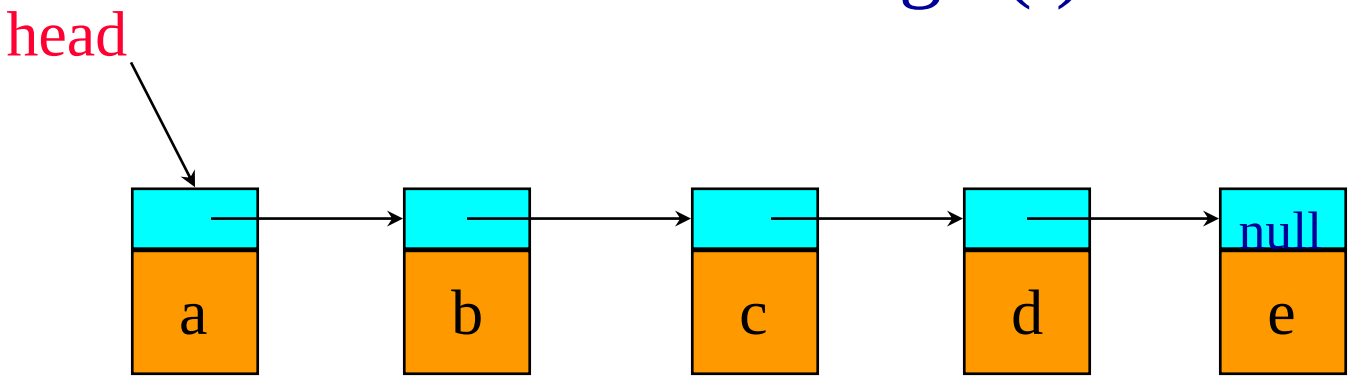
```
#return true iff the list is empty, false otherwise
def isEmpty(self):
      return self.head is None
```

The Method - size()




```
#return the size of the list
def size(self):
      size = 0;
      temp = self.head
      while(temp is not None):
            size += 1
            temp = temp.next
      return size
```

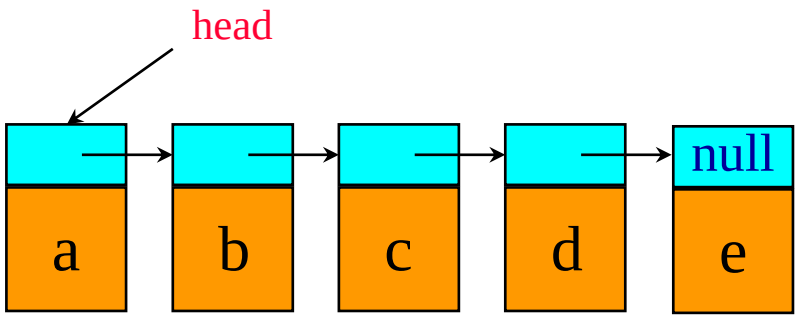
The Method – get(i)



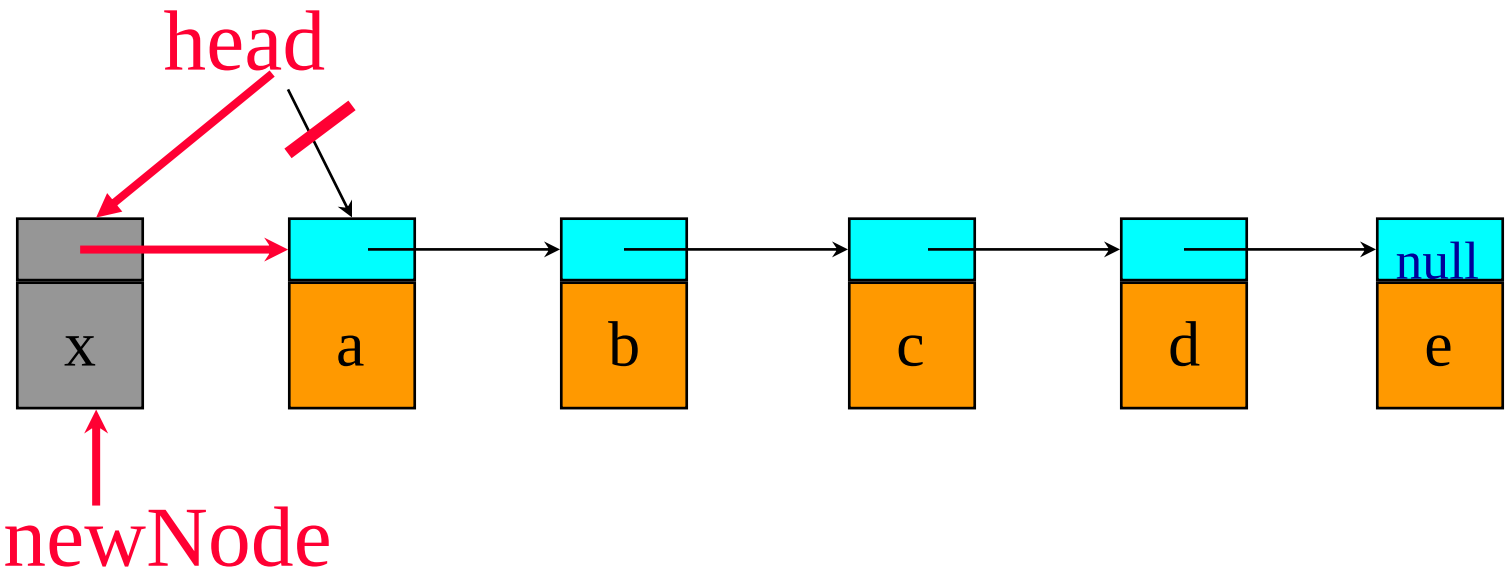

```
#return the i th element of the list
def get(self, i):
      elindex = 0
      temp = self.head
      while temp and elindex != i:
          temp = temp.next
          elindex += 1
      if temp is None:
             return None
      else: return temp.element
```

The Method – indexOf(el)

```
#find the index of a node
 def indexOf(self, el):
      temp = self.head
      tempindex = 0; #index of temp node
      while temp and temp.element != el:
          #move to the next node
          temp = temp.next
          tempindex += 1

      #make sure we found the matching element
      if temp is None:
            return -1
      else: return tempindex
```

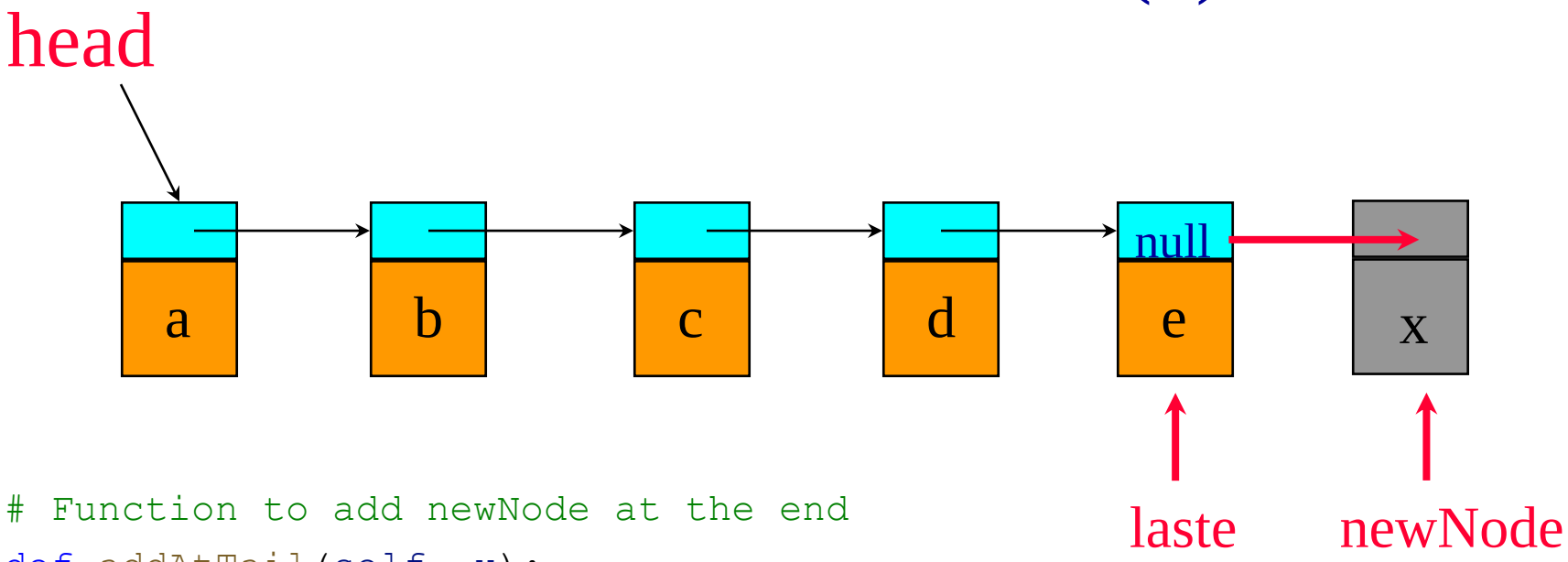


The Method – addAtHead(x)




```
# Inserting at the Beginning
def addAtHead(self,x):
    newNode = Node(x)
    # Update the new nodes next val to existing node
    newNode.next = self.head
    self.head = newNode
```

The Method – addAtTail(x)




```
# Function to add newNode at the end
def addAtTail(self, x):
    NewNode = Node(x)
    if self.head is None: # if only one node
        self.head = NewNode
        return
    #loop to the last node
    laste = self.head
    while(laste.next):
        laste = laste.next
    laste.next=NewNode
```

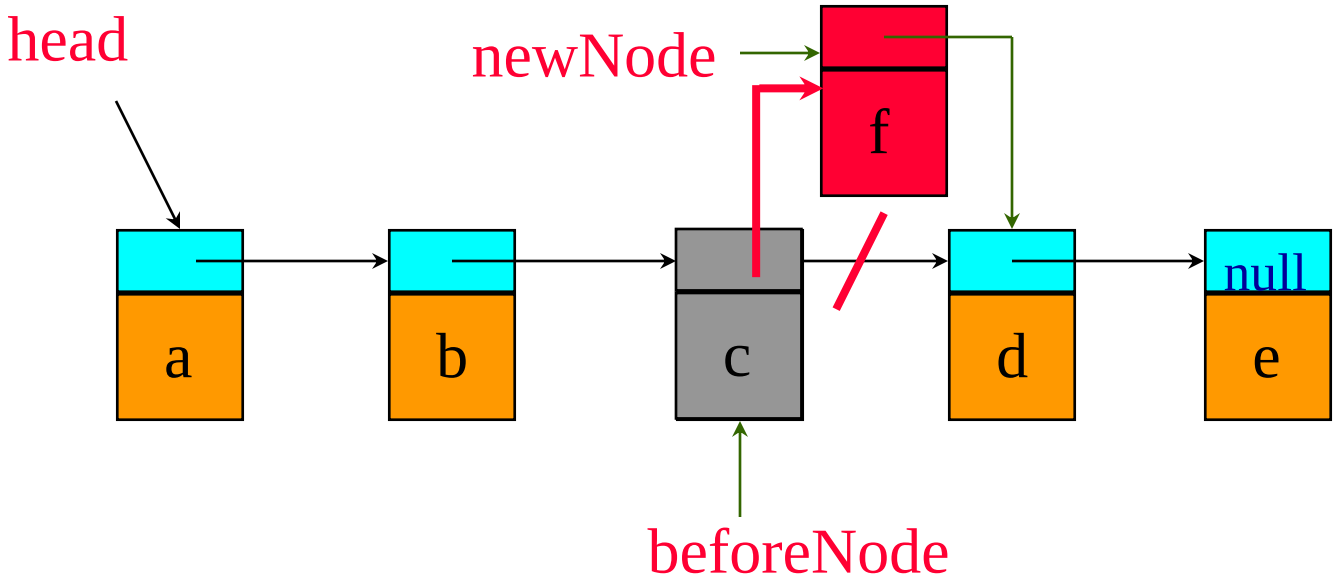
The Method – `Add(index, Element)`

Add An Element at index 0

```
#insert x as the index th element
def add(self, theIndex, x):
     if theIndex == 0:
          self.addAtHead(x)
```

The Method – `Add(index, Element)`

Two-Step add(3,'f')

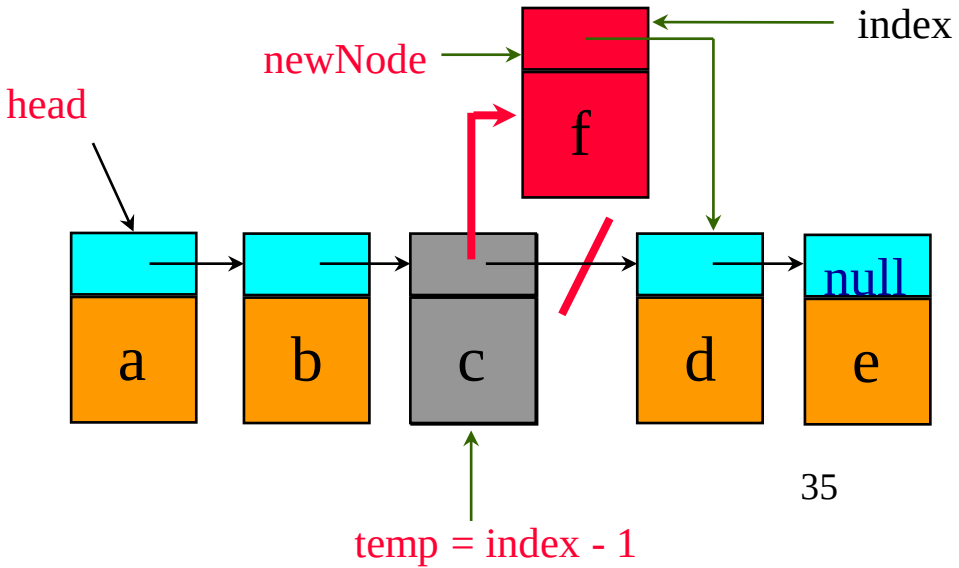



```
beforeNode = head.next.next;
beforeNode.next = Node('f', beforeNode.next);
```

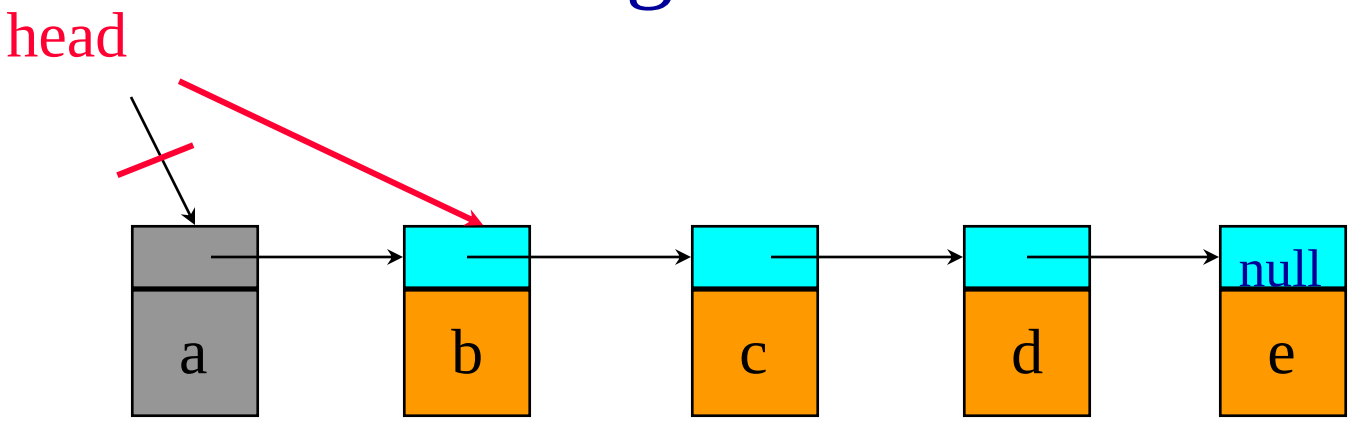
The Method – Add(index, Element)

Adding An Element

```
else:
    #find predecessor of new element
     temp = self.head
     elindex = 0
     while temp and elindex != theIndex-1:
          temp = temp.next
          elindex += 1
     temp.next = Node(x, temp.next)
```

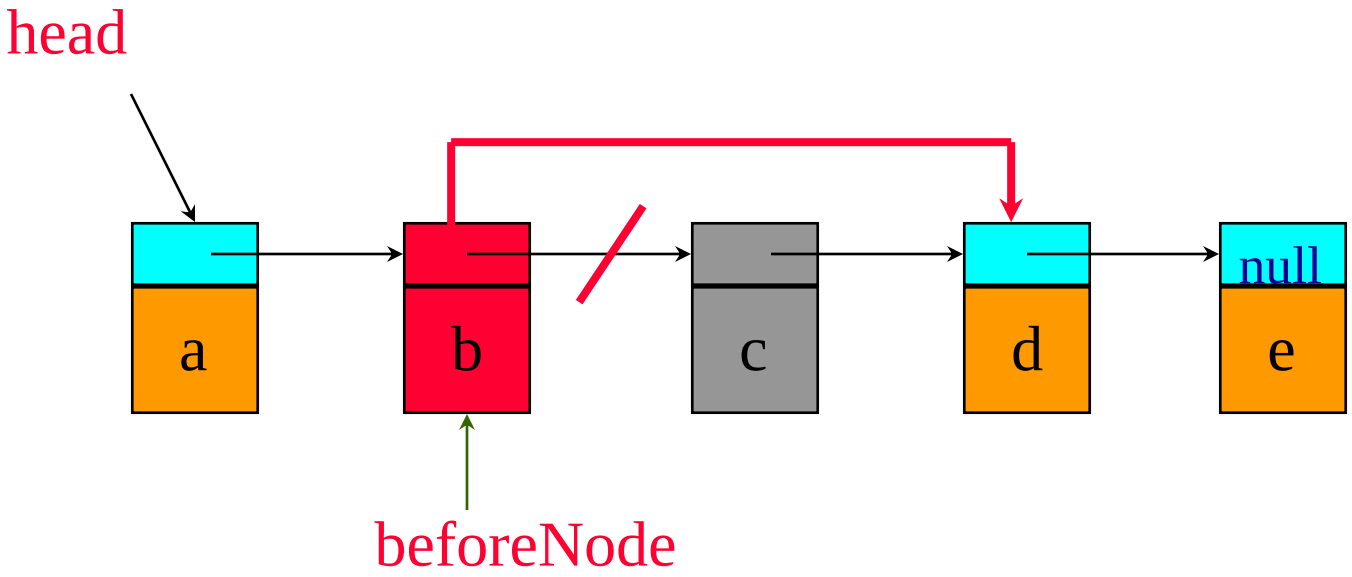


The Method – Remove(index)
Removing the first node




```
def remove(self, index):
        if index == 0:
            if self.isEmpty():
              return None
            ele = self.head.element
            self.head = self.head.next
            return ele
```

The Method – Remove(index)
remove(2)

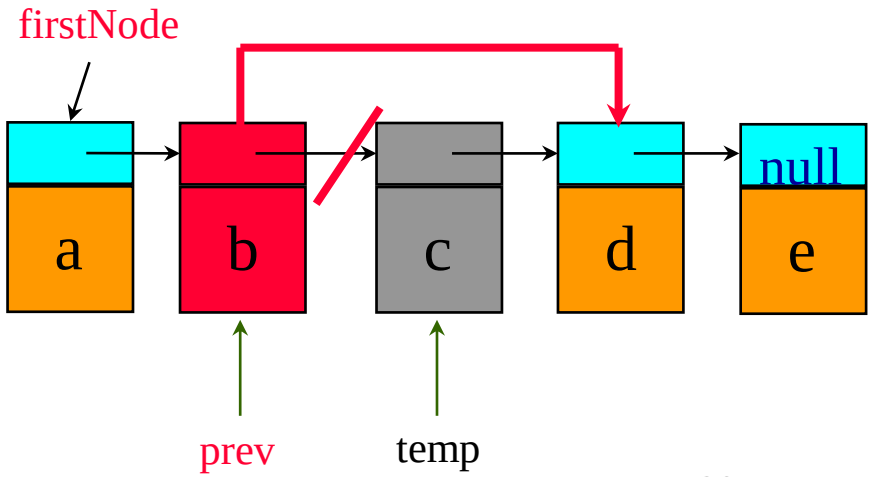


Find beforeNode and change its pointer.

beforeNode.next = beforeNode.next.next;

The Method – Remove(index)
remove(2)

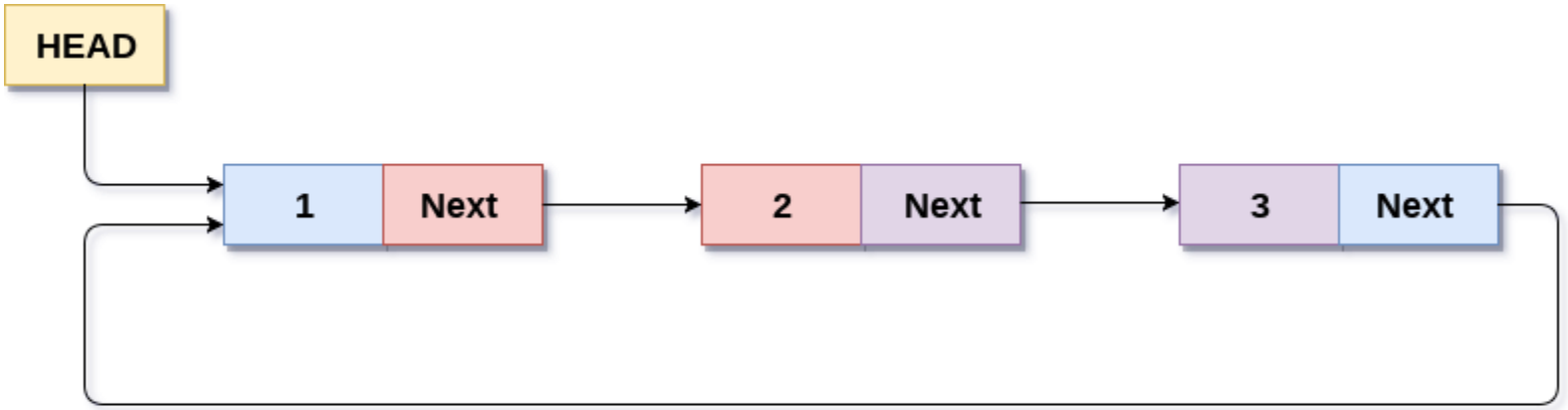
```
else:
    temp = self.head
    elindex = 0
    while temp and elindex != index:
        prev = temp
        temp = temp.next
        elindex += 1

    if (temp is None):
        return None

    prev.next = temp.next
    ele = temp.element
    temp = None
    return ele
```



Remove An Element

```
# Function to remove node with given element
def removeNode(self, Removekey):
    temp = self.head

    if (temp is not None):
        if (temp.element == Removekey): #remove the first node
            self.head = temp.next
            temp = None
            return
    while (temp is not None):
        if temp.element == Removekey: #found the node
            break
        prev = temp
        temp = temp.next

    if (temp is None): #no match node is found
        return

    prev.next = temp.next #remove the node
    temp = None
```

Circular linked lists



Circular Singly Linked List

Advantage of circular linked list

- Entire list can be traversed from any node of the list.
- It saves time when we have to go to the first node from the last node.
- Its is used for the implementation of queue.
- Reference to previous node can easily be found.
- When we want a list to be accessed in a circle or loop then circular linked list are used.

Review on List

List in Python

- The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets.
- Important thing about a list is that items in a list need not be of the same type.

List

```
list1 = ['HKCC', 'CPCE', 19, 2000]
list2 = [1, 2, 3, 4, 5 ]
list3 = ["a", "b", "c", "d"]

print("list1[0]: ", list1[0])
print("list2[1:5]: ", list2[1:5])
```

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

LList.py

More operations in List

```
print("Size: ", len(list1))
print(3 in list2)
for x in list3:
  print(x)

del list1[2]
print("After deleting value at index 2 : ")
print(list1)
```

```
Size:  4
True
a
b
c
d
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

More operations in List

```
print("After appending new item orange in the list: ")
list1.append("orange")
print(list1)

print("After Adding value HKCC at index 1 : ")
list1.insert(1, "HKCC")
print(list1)

print("Add the elements of list3 to list2:: ")
list2.extend(list3)
print(list2)
```

```
After appending new item orange in the list:
['physics', 'chemistry', 2000, 'orange']
After Adding value HKCC at index 1 :
['physics', 'HKCC', 'chemistry', 2000, 'orange']
Add the elements of list3 to list2::
[1, 2, 3, 4, 5, 'a', 'b', 'c', 'd']
```